Topics:

- **Link Lists**
  - Providing methods for operations on chains of link nodes

- **BST (Binary Search Tree)**
  - (get a head start on Wednesday’s topic)
  - BSTs are an implementation of the Map ADT from Stanford library
    - Motivation
    - Structure
    - Insert algorithm
Case 1: Removing from **front** (index 0)

Before removing element at index 0:

To remove the first node, we must change **front**.

Be sure to delete this!
// Removes value at given index from list.
// Precondition: 0 <= index < size()
void LinkedList::remove(int index) {
    ListNode* trash;
    if (index == 0) { // removing first element
        trash = front;
        front = front->next;
        delete trash;
    } else { // removing elsewhere in the list
        // left for the reader 😊
    }
    size--;
Case 2: Removing from “middle” of list (ex: index 2)

Before removing element at index 2:

<table>
<thead>
<tr>
<th>front</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

element 0

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-3</td>
</tr>
</tbody>
</table>

element 1

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

element 2

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

element 3

After:

<table>
<thead>
<tr>
<th>front</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

element 0

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-3</td>
</tr>
</tbody>
</table>

element 1

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

element 2

<table>
<thead>
<tr>
<th>data</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

element 3

Be sure to delete this!

Where should current be pointing?
How many times should it advance from front?
Case 3 (?): Removing the only element

Before:

- We must change the `front` field to store NULL instead of a node.
- Do we need a special case to handle this?

After:

Be sure to delete this!
Other list features

A nice LinkedList class will also want to have the following public member functions:

- size()
- isEmpty()
- set(index, value)
- clear()
- toString()
Binary Search Trees

Implementing the Map interface with Binary Search Trees
Implementing Map interface with a Binary Search Tree (BST)

- Binary Search Trees is one option for implementing Map
  - C++’s **Standard Template Library (STL)** uses a Red-Black tree (a type of BST) for their map
  - Stanford library also uses a BST
- Another Map implementation is a hash table
  - *We will talk about this later!*
  - *This is what Stanford’s HashMap uses*
Binary Search in a Linked List?

Exploring a good idea, finding way to make it work
Imagine storing sorted data in an array

- How long does it take us to find?
  - **Binary search!**
  - **O(logn): awesome!**

- But slow to insert
  - **Move everyone over to make space**
  - **O(n):** not terrible, but pretty bad compared to log(n) or O(1)
  - In contrast, linked list stores its nodes scattered all over memory, so it doesn’t have to move things over
Q. Can we do binary search on a linked list to implement a quick insert?

A. No.
   - The nodes are spread all over memory, and we must follow “next” pointers one at a time to navigate (the treasure hunt).
   - Therefore cannot jump right to the middle.
   - Therefore cannot do binary search.
   - Find is $O(N)$: not terrible, but pretty bad compared to $O(\log n)$ or $O(1)$

Binary Search Tree can be thought of roughly like a linked list that has pointers to the middle, again and again (recursively), to form a tree structure
An Idealized Binary Search Tree
TreeMap

This is basically the same as Stanford Map. Here in class we’ll call it TreeMap just to be explicit about its implementation.
tree-map.h

template <typename Key, typename Value>
class TreeMap {
public:
    TreeMap();
    ~TreeMap();

    bool isEmpty() const;
    int size() const { return count; }
    bool containsKey(const Key& key) const;
    void put(const Key& key, const Value& value);
    Value get(const Key& key) const;
    Value& operator[](const Key& key);

    //...(continued on next page)
```cpp
// class TreeMap continued...
private:
    struct node {
        Key key;
        Value value;
        node* left;
        node* right;
    };
    int size;
    node* root;
};
```
BST put()

Pretty simple!

- If key > node’s key
  - Go right!
- If key < node’s key
  - Go left!
- If there is nothing currently in the direction you are going, that’s where you end up

- Example: put(23)